Extended AIML-Based Interactive Dialogue System for Social Robots

Saxon M.S.[1], Ge S.[2], Wang C.[3], Liu X.[4]

Social Robotics Lab, National University of Singapore

ABSTRACT

Natural language processing is a current frontier in artificial intelligence. Several approaches exist to tackle the problem of writing algorithms that can interpret natural human language input. One approach, known as AIML, (Artificial Intelligence Markup Language) was developed to create the ALICE bot, an NLP "chat bot" that can carry basic conversation with a human user over the internet. AIML is an XML-based language that, on the highest level groups possible conversation topics and then uses recursive pattern matching principles and context to pick a pre-written response to an input. The main task of this project centered around creating a fork of the open-source PyAIML Python AIML specification to enable setting robot-specific variables natively inside the AIML source, including A*STAR Abacus Speech Recognition-based microphone input for users to talk to the robot, and adding a "modifyaiml" function that allows the robot to change its own code to correct errors and add information.

INTRODUCTION

Over the course of this project, multiple different natural language processing systems were tested for use in social robotics applications, principally, AIML by the Artificial Intelligence Foundation and the Apollo Dialogue Management System by the Singaporean Agency for Science, Technology and Research (A*STAR). Additionally, the Abacus Speech Recognition system from A*STAR was used to add microphone input capabilities.

AIML

AIML was created by Dr. Richard S. Wallace in 2001 to create the Artificial Linguistic Internet Computer Entity (ALICE Bot), a web-enabled chat bot that uses recursive pattern-matching to pick a response to an input. It is based on XML and uses tags such as "`<topic>`", "`<pattern>`", and "`<template>`" to denote sections of text within a conversation topic, patterns to be matched, and response templates respectively. PyAIML is a Python AIML interpreter created by Cort Stratton that implements AIML version 1.0.1

[1] Undergraduate Researcher
[2] Principal Investigator
[3] Graduate Student
[4] Graduate Student

completely in vanilla Python 2.X. The PyAIML can either directly load ("learn") AIML files on every instantiation or it can load a precompiled binary "brain" of AIML data. Brain files load much faster than AIML text files, particularly for larger databases of responses, since they do not require parsing. However, they cannot be easily "un-compiled" back into AIML for direct human editing. A brain file can also be directly modified by the kernel, in-conversation, to add new functionality and responses, AIML files cannot be modified this way in the base version.

Apollo Dialogue Management System

Apollo is a much newer system than AIML. Developed over the last three years by A*STAR researchers, it is more purpose-built for the kinds of social robotics applications pertinent to this project than AIML is. Rather than being a language system, Apollo is a "dialogue management system," a Windows desktop application that combines an AIML-like XML specification for conversations with several included DLL plugins that enable network functionality, text-to-speech, and Python code execution.

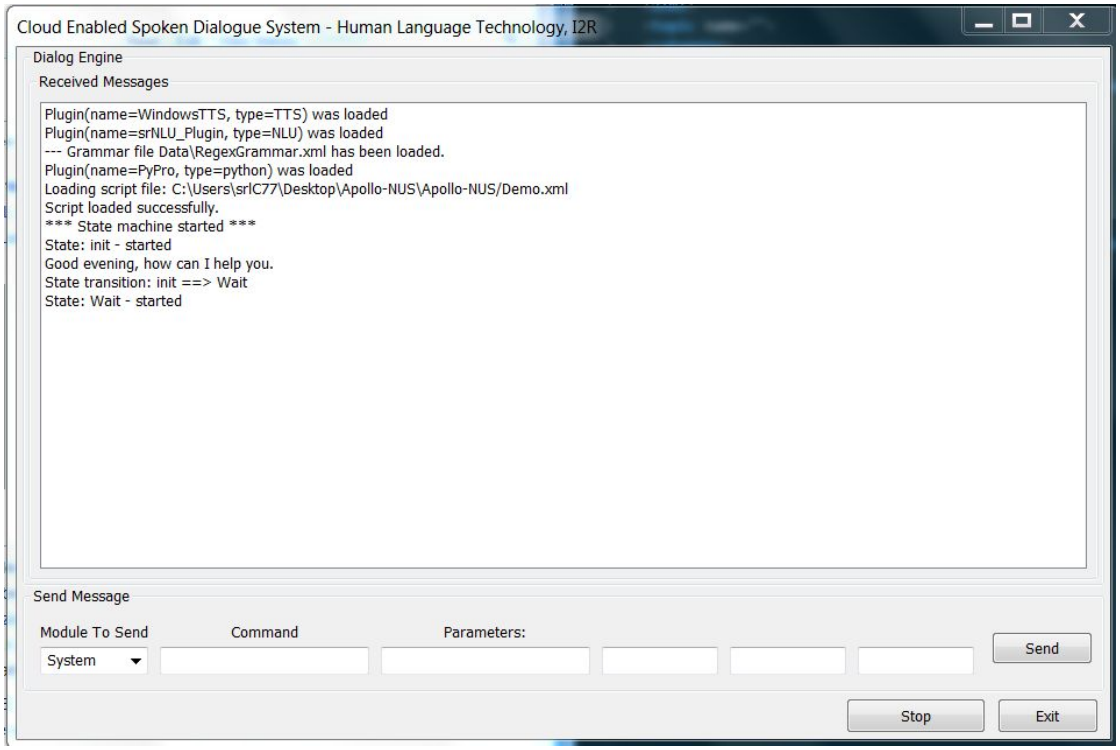

Figure 1. The GUI of the Apollo Dialogue Management System

While the Apollo system is much more full-featured than vanilla AIML, it was considerably more difficult to work with on this project. Since A*STAR decided to create its own scripting language for its natural language processing system, and since said system is still a work in progress, full, up-to-date documentation proved difficult to find. This led to many roadblocks on setting up simple tasks and forced the move to a fully AIML-based system. However, with further updates by A*STAR, and with more available time the Apollo system is

worth a second look, in particular thanks to its built-in frame-based dialogue management which allows for very easy form-filling application setup, a functionality AIML lacks.

Abacus Speech Recognition

The Abacus Speech Recognition (ASR) system was also created recently by A*STAR to allow for internet speech recognition. Using microphone input, a Python code sends "chunks" of recording to the A*STAR server for interpretation and the server returns recognized text. Because an ASR implementation can be made using just vanilla Python, adding ASR functionality to other codes discussed in this paper was very straightforward.

PROCESS

At first, work was done to integrate the Apollo DMS system and ASR speech recognition. This was fairly straightforward, using a Python program that would send microphone input to the ASR server, and then encode the returned text into a JSON string, to be sent to the DMS application running on the same computer as a TCP/IP message.

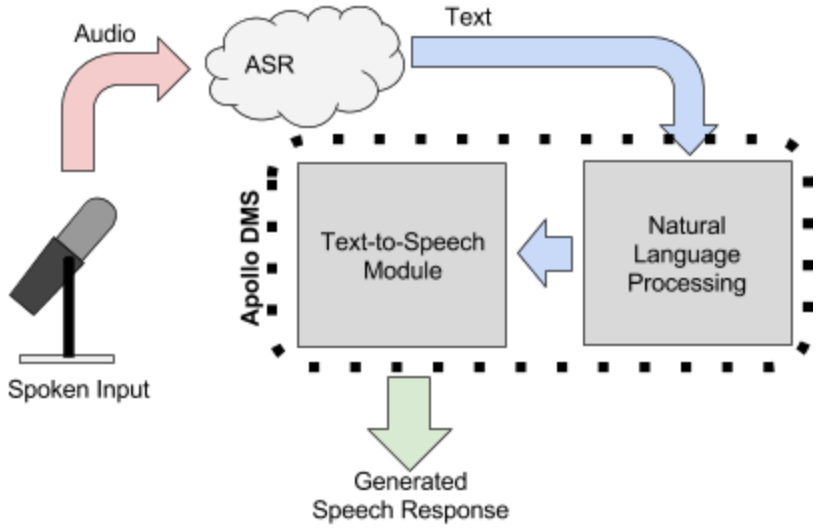

Figure 2. How the Apollo DMS system works with ASR code

Due to the sparse documentation, it proved time consuming to figure out things like how to format the JSON messages and how to configure the DMS to receive and interpret the information correctly. During lulls in production resulting from these difficulties, functions and classes were developed to make the process of setting up the system easier.

Abandoning Apollo DMS

After a few weeks of trying to work with the Apollo Dialogue Management System, the decision was made to look for another NLP platform to work with. In many respects, AIML and the Apollo XML spec are very similar. Thus, AIML was an easy choice.

At this point, the question became what changes to make to the AIML system. It is optimized for the design of textual internet chat bots. Because social robots have a wide range of designs, extensibility of responses was added. In vanilla AIML, the responses a chat robot makes to inputs is not extensible, returning only a string in response to an inputted text pattern. An updatable class was substituted in the extended version of the kernel, so that functionality such as gestures, emotions, or even images, web links, or function calls can be added to the AIML output.

The final functionality added to the extended AIML was "self-editing." In vanilla AIML, a kernel loads AIML script on instantiation for its rule set, or loads a compiled binary AIML ruleset as a "brain file." Brain files can be edited by the kernel as it runs by being told "bad answer" and given the appropriate command. However, these brain files cannot be converted back into AIML or edited by a human using a text editor. Thus a feature was added to rewrite AIML files in the Python code, making the AIML files themselves more "brain-like."

DESIGN

Speech Interpretation Functions

A set of sample functions for use of ASR had already been provided, including one that starts recording the user talking when called and records for a pre-set amount of time. This doesn't very accurately resemble natural conversation, so one of the early goals was setting up a recording function that waits until prompted to start recording, and naturally stops.

The first such code would repeatedly send 10-second recordings to the ASR server. Once the user said the phrase "OK Computer," the program would receive "`OK COMPUTER`" from the ASR server and begin recording the message. This approach had many problems. First, it was very inefficient, wasting a lot of ASR server calls. Second, saying "OK Computer" to begin a prompt is even less life-like than just a strict timed recording. Finally, the ASR recognition can make errors, particularly with the low quality microphone that was being used at the beginning of the project, so even when the user would say "OK, Computer," the recording would often not activate. In light of these errors, this function was scrapped.

The second idea was to wait until some defined length of sound louder than the microphone noise floor was detected, and then begin the normal ASR process. However, if this

length is too short, background noises trigger the microphone, and if it is too long, the beginning of the phrase being said by the user is cut off, leading to an incorrect interpretation of the user's phrase. Thus, this script was scrapped as well.

The function that was ultimately developed to solve this problem is called "dynamic record." In dynamic record, two ring buffers are implemented using a double-ended queue from Python's collections module. The first is filled in step (a), where the ring buffer constantly records new "chunks" of audio data and dumps the old ones. Once the client detects a specified length of audio input above the noise floor, it dumps the contents of the ring buffer into the ASR server, and then switches to the same regular recording mode from the first sample code. As this recording continues, ring buffer (b) is used to determine how long the last silence was. Once a ring buffer is filled with silent chunks (no detected speech) the process stops and the function returns the text retrieved from the ASR server.

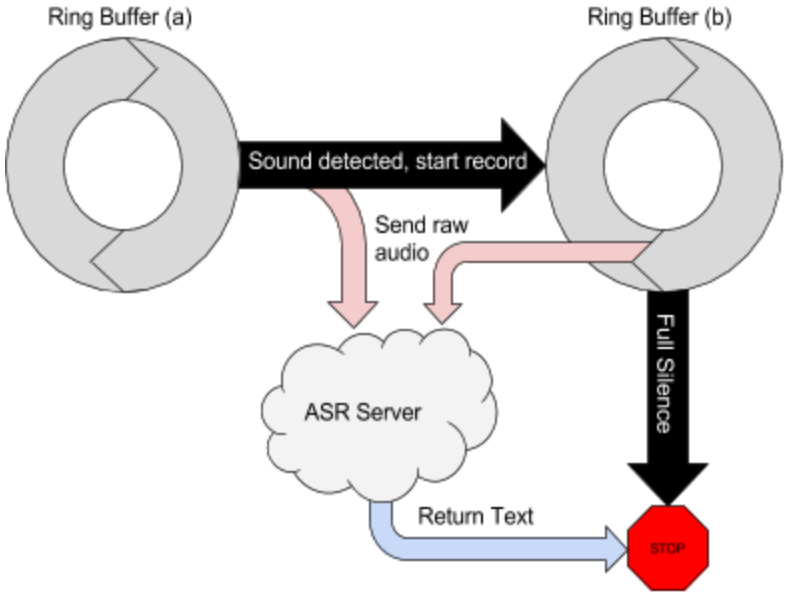

Figure 3. Diagram showing how dynamic record works

The dynamic record function more closely approximates the experience of being listened to by a human, as it waits for its conversation partner to speak, and listens to the whole comment rather than cutting them off during a long sentence as the other methods do.

AIML Improvements

After the switch to AIML discussions were made about how to improve its functionality for social robotics applications. As a result the two big AIML kernel improvements were chosen as detailed above.

Non-Text Output in AIML

Many social robots implement supplemental features for conversation such as hand gestures, facial expressions, and the like. However, the vanilla PyAIML kernel outputs plaintext strings in response to queries. AIML already has the capacity to store variables as "predicates," but accessing them through "`kernel.getPredicate(name, sessionID)`" calls is rather cumbersome. Thus, the decision was made to change the return type of the "`kernel.respond(message, sessionID)`" call to a custom class contained in the kernel class. By default this class contains a string, "message," which contains the text response that respond used to return alone, and another string, "emotion," which can be set directly as the "emotion" predicate in the AIML.

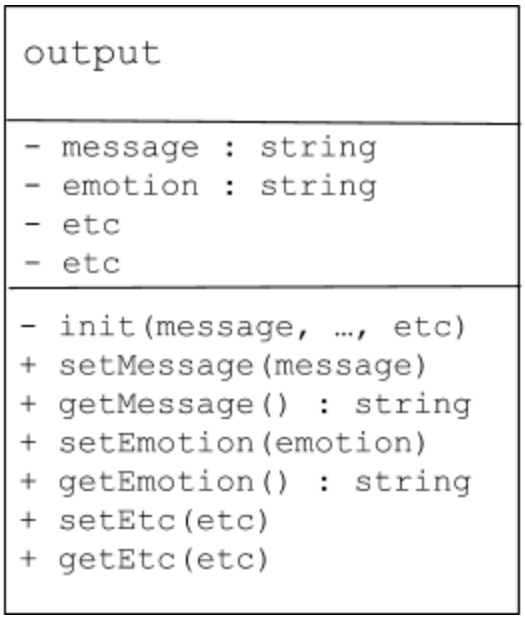

Figure 4. UML class diagram for output type

Setting emotion, or any other of these supplementary output types in the AIML is very straightforward, requiring only a pair of "set" tags.

```
<category>
<pattern>Hello, my name is *</pattern>
<template><think><set name="emotion">HAPPY</set></think>Nice to meet
you, <star index="1"/></template>
</category>
```

Using this data in Python robot control code is seamless. Once an output object is returned from "`kernel.respond(message)`" call, any of the "get" functions described in the UML diagram can be called to get the appropriate robot control data. This extended Python AIML specification can be used to control all kinds of robot functions natively within the AIML.

AIML Source Editing in the Kernel

As has been mentioned before, the AIML kernel is already capable of editing its own pre-compiled brains. However, once these brains are compiled the rules added to them cannot be changed. Emotion functionality as discussed above can no longer be edited and the "brain files" appear meaningless to a user when opened in a text editor. To facilitate on-the-fly learning while maintaining the editability of AIML files, an "`aimlmod`" function was created.

Aimlmod is very simple. It requires that every conversation topic have its own dedicated aiml file and that they be organized as follows: the empty conversation topic, "" be in the file named "basic chat.aiml" in the root directory of the program, and all other topics "xyz" be at the location "/topics/xyz.aiml." Files have to have no indentation. When these conditions are met, aimlmod checks the current topic when called and appends the new given rule to the topic's aiml file. This functionality can potentially allow, for example, the robot to ask users how to correctly answer one of their own queries to it, and then update its database on the fly, while still leaving the database editable by software designers.

CONCLUSIONS

In this new extended specification, robot designers can add emotion, face, and hand gesture outputs directly in the AIML code to the textual pattern responses, making enhanced lifelike behavior simpler to implement than in vanilla AIML. Additionally, using A*STAR's Abacus Speech Recognition system, this communication system can directly interpret spoken English input through a connected microphone. Finally, an AIML modification system was added, wherein the user can notify the robot of a "bad answer" and the robot edits its own AIML file, saving the modification, and then reloads it and continues on with the correct response in its system. These enhancements to the AIML system allow robot designers to enable more life-like communication for conversational social robots.

Upon returning to Arizona State University in the fall I hope to continue work improving this system, and possibly eventually releasing an improved fork of Python-based AIML to the public.

ACKNOWLEDGMENTS

Special thanks to Wang Chen and Liu Xiaomei for getting me started in the lab and helping me understand the software. Thanks to Dr. Jiang Ridong and Dr. Lim Boon Pang from A*STAR for help with using the Apollo DMS and Abacus Speech Recognition software, respectively. Thank you to Dr. Sam Ge for welcoming me to the lab and taking me on as a SERIUS student. Thanks again to Wang Chen and to Li Mingming for a good time hanging out in the lab, and all the lunch trips and fun conversations we have had during this summer.